# FormalCORE™ PCI/32
# A Formally Verified VHDL Synthesizable PCI Core

Bhaskar Bose, M. Esen Tuna and Ingo Cyliax
Derivation Systems, Inc.
Carlsbad, California, USA.
www.derivation.com

**Abstract**

This paper describes an integrated design methodology for the use of formal methods with existing tools in the context of developing FormalCORE PCI/32. The primary goal is to develop technology for the design and verification of formally verified IP cores that includes all the features, documentation, and support necessary to insure integration into designs with the high degree of reliability provided by the application of formal methods. Validation techniques used in developing these cores include formal specification, formal synthesis, simulation, hardware emulation, theorem proving, and model checking.

## 1 Introduction

The PCI[6,7] Local Bus is a high performance, 32-bit or 64-bit bus with multiplexed address and data lines. The bus is designed for use as a high-speed interconnect mechanism between peripheral components and processor/memory subsystems.

FormalCORE™ PCI/32 is a synthesizable VHDL[4] 32-bit, 33MHz PCI interface core targeted to programmable hardware. The VHDL description is formally synthesized using our DRS[1,2] formal synthesis system and formally verified using the Verysys PropertyProver model checker to be compliant with the v2.1 PCI specification.

The overall goal of the project is increased assurance by using a variety of formal methods technologies in concert to attack a practical problem. We have developed the methodology for the design and validation of VHDL cores with a variety of tools that can serve as documentation, and increase assurance. In meeting the primary goal of the project we achieve a reduction in the development time as well. Once the design flow was in place, correcting specification bugs and rechecking the properties was a routine task rather than a challenge.

A key benefit to this approach is that it allows for the deployment of formal methods into current engineering practice via pre-designed, pre-verified components that meet the stringent reliability and safety requirements that are necessary in avionics and space applications. These components can then be integrated into larger designs providing the building blocks for complex designs and the foundation for design reuse.

In developing the FormalCORE technology we rely heavily on both formal and traditional design and verification tools. We recognize at the early stages of planning that a comprehensive approach to the integration of formal verification techniques to an existing design flow is critical to the success of the technology. A well implemented design and verification strategy, incorporating formal techniques at key points in the design flow minimizes the likelihood of design errors.

## 2 The PCI Bus Protocol Standard Revision 2.1

The PCI bus specification was first developed by Intel Corporation and was released in June 1992. It was intended to define an industry standard for local bus architectures. Revision 2.1 became available in early 1995 and is managed by a consortium of industry partners known as the PCI Special Interest Group. The specification is a 282-page English language document describing the protocol, electrical, mechanical, and configuration specification for PCI components and expansion boards.

The PCI specification defines two possible PCI agents, a *master* and a *target*. The master is the device that initiates a transfer. The target is the device currently addressed by the master for the purpose of performing a data transfer. The master and target state machines are independent. However, a master device must incorporate a target device for the purpose of responding to system configuration requests.

The minimum PCI compliant device satisfies the requirements of a target-only device. This device requires 47 pins and can only respond to a master initiated transaction. A master device requires two additional signals, (REQ# and GNT#), for it to handle data and addressing, interface control, arbitration, and system functions. Figure 1 illustrates the required and optional signals for a PCI compliant device. The signals on the left are required pins for target and master devices. The signals on the right are optional pins and are used to support the 64-bit extension to the specification, exclusive access (LOCK#), interrupts, cache support, and the JTAG (IEEE 1149.1) boundary scan interface.

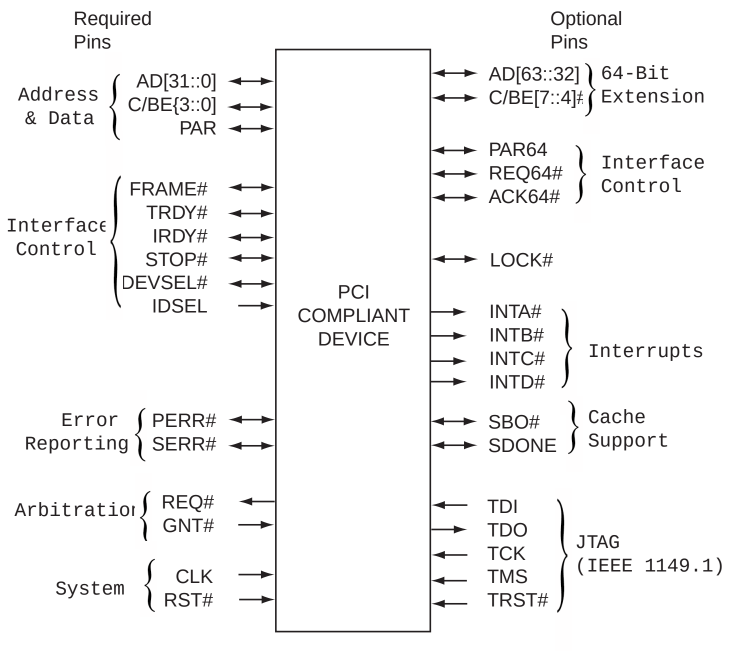

Figure 1: PCI Compliant Device Signals

The heart of the PCI Bus Protocol is the burst transfer mechanism. A burst transfer consists of a single address phase followed by two or more data phases. The start address and transaction type are issued during the address phase. The target device latches the start address into an address counter and is responsible for incrementing the address from data phase to data phase. Figure 2 illustrates a sample read transaction.

A basic bus cycle involves the FRAME#, IRDY#, TRDY#, C/BE# control signals as well as the multiplexed address/data AD[31:0] lines and the parity signal PAR and DEVSEL#. The bus cycle starts with an *address phase*. This is the first clock after FRAME# is asserted by the master. During this cycle, the address lines carry the desired address and the C/BE# signals the bus command. Bus commands encode the address space and direction of transfer. There are also some special bus cycles, like interrupt acknowledge and various memory transfer modes. After the address phase, the master goes into the *data phase*.

The addressed target, will then decode the address to determine if it needs to take the bus cycle. It can decode either as a fast/medium/slow decoder, which are 1,2,3 cycles after the address phase. Once it has decoded and accepted the bus cycle, it asserts the DEVSEL# signal to signal that it will take the bus cycle. When the master has sent data via the AD[31:0] or when it is ready to receive data, it will assert the IRDY# signal. The target indicates its readiness with the TRDY# signal. Only when the TRDY# and IRDY# signals are both asserted, will a data transfer take place. Otherwise wait states are inserted. The master controls how much data is

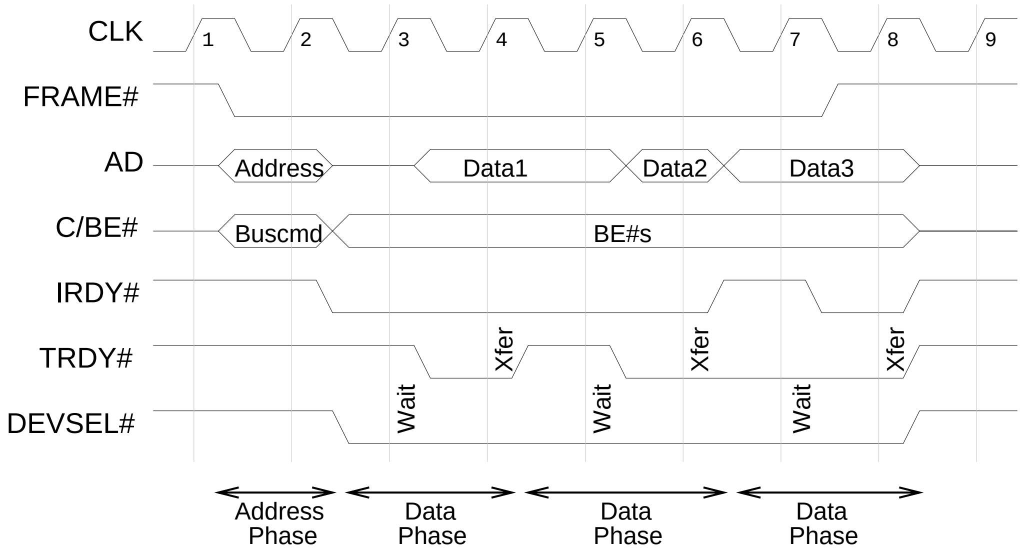

Figure 2: PCI Timing Diagram

transferred. When it is done transferring data, it will de-assert FRAME# on the last data phase. When the target sees neither FRAME# or IRDY#, the master has finished.

The target uses the STOP# signal to signal the master that it has to terminate the current transaction. The PCI Target asserts combinations of TRDY#, DEVSEL#, and STOP# to signal different termination conditions. The PCI protocol is specified in plain English. The specification contains rules such as:

*"Data is transferred when IRDY# and TRDY# are asserted."*

*"When either TRDY# or IRDY# is deasserted, a wait cycle is inserted and no data is transferred."*

## 3 FormalCORE PCI/32 -- A Formally Verified PCI Interface

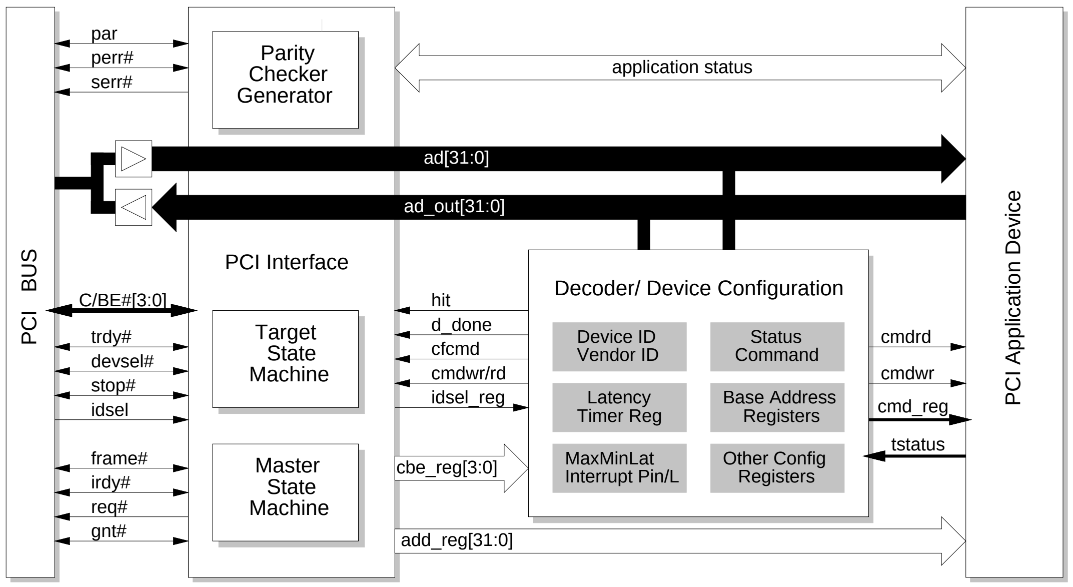

Figure 3: FormalCORE PCI/32 System Architecture

FormalCORE PCI/32 is a synthesizable VHDL 32-bit, 33 MHz PCI interface targeted to programmable hardware, formally verified to be compliant with the v2.1 PCI specification. Figure 3 is a block diagram of the FormalCORE PCI/32 system architecture.

The design is composed of three primary modules. A PCI Interface Module, Decoder/Device Configuration Module, and PCI Application Module. The PCI Interface Module is the primary interface to the PCI bus and user application. It contains the Target and Master state machines, parity circuit, and implements the bus protocol. The Decoder/Device Configuration Module contains the PCI configuration registers and address decode circuitry. The PCI Application Module is a stub module defining a backend interface. This module is used to integrate the user's application into the PCI core. It is not specified in v2.1 since it is dependent on the specific device. For example, the Application Interface would vary widely between a video device and a modem. This partitioning allows us to swap different application backends to the existing core with minor modifications.

## 4 Design and Verification Tools

The software tools comprising our design and verification suite included:

- DRS (Derivational Reasoning System), formal synthesis system from Derivation Systems, Inc. to develop high-level formal behavioral specification, high-level simulation, hardware emulation, and formal synthesis to VHDL and gate-level netlist. We use DRS to derive a structural specification from the top-level behavioral description, synthesize VHDL code and PVS theories. The system was also used for functional simulation of the top-level specification, and as the interface to hardware emulation of the synthesized design.

- PVS[5] (Prototype Verification System) from SRI for validating safety and liveness properties of the top-level behavior specification.

- Verysys PropertyProver[8] and StructureProver[8]. PropertyProver is a state-of-the-art model checker that can verify model properties at the Behavior, RTL and Gate levels. StructureProver is a high-performance, high capacity equivalence checking tool that can be used at the RTL and Gate levels. The Verysys tool suite was chosen for its support of the IEEE 1076 VHDL standard and hierarchical verification. In addition, PropertyProver generates an input sequence and a VHDL testbench for counter-examples. The built in VHDL simulator can be used to simulate the counter example.

- Verysys Circuit Interface Language[3,8] to formally describe circuit properties. These properties are described using temporal relationships between the various input and output ports of the circuit. CIL is used to describe the PCI Compliance Model to validate the VHDL core. Properties are written in an assumption-commitment style. Predicates in the logic are written using VHDL syntax.

- ModelSim from Model Technologies for VHDL simulation. ModelSim is chosen because it is a full featured VHDL simulator providing accurate modeling of the language. It provides a rich set of features.

- Xilinx Foundation Express[9] for VHDL synthesis, gate-level timing analysis, gate-level simulation, and FPGA programming. Foundation Express incorporates the Synopsys Express VHDL compiler and Aldec gate-level timing analyzer and simulator. Foundation Express provides a low-cost, comprehensive solution for FPGA programming. The entry to the tool can be VHDL, Verilog, Schematic entry, or gate-level netlist. Xilinx offers a variety of chips that are PCI compatible and is an industry leader in programmable hardware.

## 5 Design and Verification

The primary design criteria for FormalCORE PCI/32 was to synthesize a VHDL model from DRS that would run at 33Mhz, optimized for size, and compatible with the various VHDL level tools. The generated VHDL had to be compatible with the Verysys model checker, Synopsys FPGA Express compiler, and Model Technologies VHDL simulator.

From the PCI Specification document, we developed a formal PCI compliance model in CIL, Verysys circuit interface language. These properties are described using temporal relationships between the various input and output ports of the circuit. They are extracted from the PCI rules in the specification document.

Formal design and verification is a theme that runs throughout the lifecycle of the FormalCORE PCI/32 development. Verification tools were used continuously once the design reached a state where the tools were applicable. DRS synthesis served as a backplane for the design flow. Changes in the design were reflected in the DRS top-level specification and the VHDL was re-synthesized.

The need for verification in this project was two fold. First the specification had to be proven to meet the PCI specification properties. The correctness of the specification in derivation is assumed, not proven. Secondly, even though DRS guarantees correctness of its transformations in the original specification, the state representation and the VHDL translation are not reasoned about. Therefore, the generated VHDL had to be shown to satisfy the same properties as the initial DRS specification.

Once a stable DRS specification was established, PVS was employed to validate the DRS top-level description. DRS was then used to derive a structural description from the top-level specification and generate VHDL. Verysys model checker, Model Technologies VHDL simulator, and Synopsis VHDL compiler were used for VHDL property verification, simulation and synthesis. The synthesized gate-level design was simulated with the Xilinx simulator.

Several modes of validation were always running in parallel. We performed functional simulation of the top-level and structural DRS descriptions. We simulated the design both at the VHDL and gate-level. Formal verification at the high-level, and formal verification at the VHDL level were used to validate properties of the design. The design flow (Figure 4), from high-level formal specification to running hardware can be characterized as five stages of design.

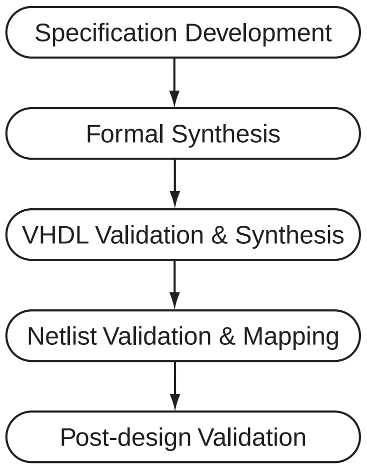
Specification Development
Formal Synthesis
VHDL Validation & Synthesis
Netlist Validation & Mapping
Post-design Validation

Figure 4: Design Flow

The design flow reflects a top-down design methodology. It provides for the formal specification and verification at an abstract behavioral level, and a systematic process to refine the design to a concrete VHDL implementation. The design flow incorporates formal and traditional validation techniques. The use of DRS and formal methods contributes to the soundness of the specification and implementation, and VHDL provides an industry standard language to interface to other tools. Figure 5 details the design and verification flow and the tools used. Shaded boxes denote formal tools. Shaded ovals denote formal specifications. Clear boxes denote traditional design tools.

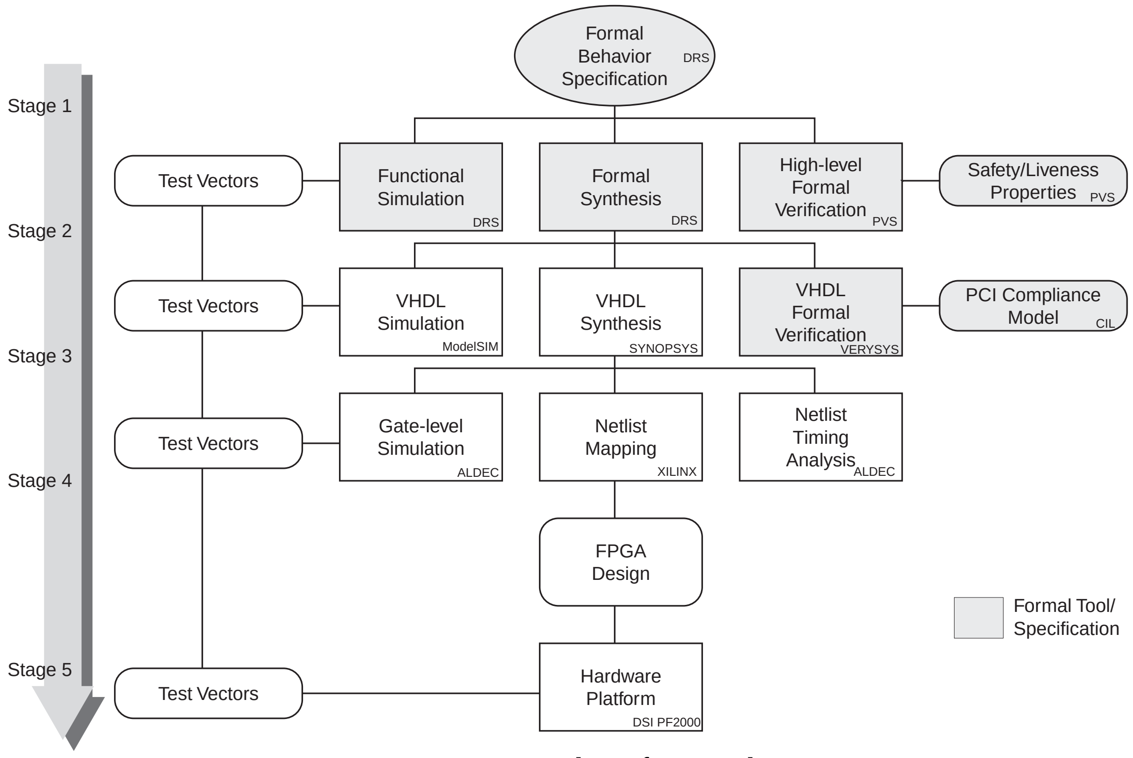
Formal Behavior Specification DRS
Stage 1
Stage 2
Stage 3
Stage 4
Stage 5
Test Vectors
Functional Simulation DRS
Formal Synthesis DRS
High-level Formal Verification PVS
Safety/Liveness Properties PVS
VHDL Simulation ModelSIM
VHDL Synthesis SYNOPSYS
VHDL Formal Verification VERYSYS
PCI Compliance Model CIL
Gate-level Simulation ALDEC
Netlist Mapping XILINX
Netlist Timing Analysis ALDEC
FPGA Design
Hardware Platform DSI PF2000
Formal Tool/ Specification

Figure 5: Design and Verification Flow

## 5.1 Specification Development

In the first stage the top-level behavior specification is developed and validated using simulation and formal verification. Verification begins early using the DRS functional simulator. A high-level behavioral model is written in DRS and run against test vectors. This behavior model becomes the reference model for all subsequent verification and synthesis.

```
[b_busy
 (lambda (add_reg cbe_reg idsel_reg ...)
   (let ([devsel_lo_o HI]    [serr_lo_o HI]    [trdy_lo_o HI]
         [stop_lo_o (not (and (or t_abort term)
                              (or wrcmd (and rdcmd tar_dly))))]
          ...)
     (if (and (or frame (not d_done)) (not hit))
         (b_busy ...)
         (if (and (or frame irdy)
                  (and hit (and (or (not term) (and term ready))
                           (or free (and locked l_lock_lo)))))
             (s_data ...)
             ...))))]
```

Figure 6: Code fragment for Target Interface b_busy state

The top-level DRS specification is a collection of communicating state machines. Each state machine is defined in terms of a set of mutually recursive function definitions. A fragment of the b_busy state of the Target Interface is depicted in Figure 6. Because of the reactive nature of the protocol specifications, the specification is written at a fine level of granularity. The specification captures the complete synchronous behavior of the PCI core circuit.

DRS descriptions were written for the master and target state machines along with their lock machines, the configuration/decode circuit, the parity circuit, and a basic application backend. The chip-level glue-logic was also written integrating all the modules into a single core. Figure 7 illustrates the modules and their interconnectivity.

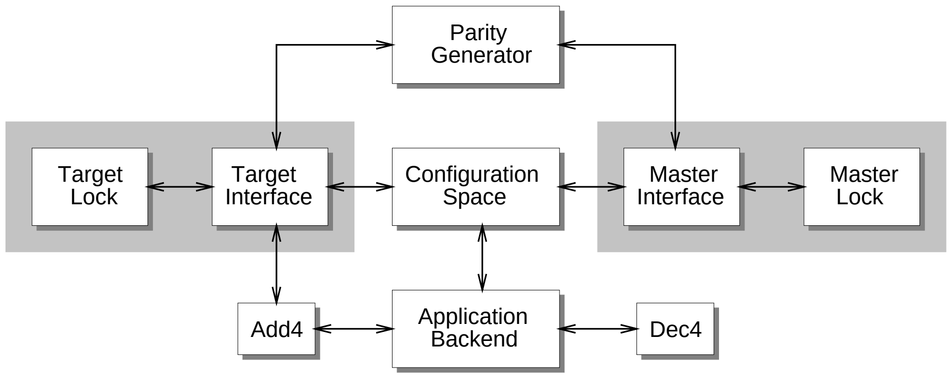

Figure 7: DRS Specification Hierarchy

An abbreviated form of the top-level DRS description is shown below. The module instantiations are show in bold.

```
(define mchip
  (lambda (cbe_lo ad par idsel frame_lo irdy_lo trdy_lo stop_lo lock_lo
           perr_lo serr_lo devsel_lo gnt_lo)
    (stream-letrec
        ([tsbuf  (lambda (o oe) (if oe o #\z))]
         [parity (lambda ((d0 ... d31) (c0 c1 c2 c3)) (b-xor ...))]
         ...)
      (letrec (...) ;; -- Component descriptions
  (system-letrec
    ([(add_reg cbe_reg idsel_reg ...) (target_xface ad cbe_lo par ...)]
     [(conf_data hit d_done cfcmd ...) (target_conf add_reg ad ...)]
     [(ad_o cbe_lo_o tstatus t_abort ...) (backend mxfer add_reg ...)]
     [(lock_lo_oe own_lock ...) (master_xface par idsel frame_lo ...)]
     [(tfree tlocked) (target_lock frame_lo lock_lo l_lock_lo hit ...)]
     [(lock_free) (master_lock frame_lo lock_lo own_lock)]
     [par_c (parity (if par_dir ad_out ad)
                    (if par_dir cbe_lo_out cbe_lo))]
     [ad_out (tsbuf32 (if (b-or ior cmdwr) ad_o conf_data)
                      (b-or ad_oe m_ad_oe))]
     [frame_lo_out (tsbuf frame_lo_o frame_lo_oe)]
     [irdy_lo_out (tsbuf irdy_lo_o irdy_lo_oe)]
     ...)
  (list ad_out cbe_lo_out par_out frame_lo_out trdy_lo_out irdy_lo_out
        stop_lo_out perr_lo_out serr_lo_out devsel_lo_out req_lo
        lock_lo_out ...))))))
```

The DRS behavior model is automatically translated into a PVS theory to perform formal verification. The primary goal is to verify that the specification satisfies a set of high-level safety and liveness properties. Inconsistencies in the top-level specification found by PVS are then manually corrected in the DRS specification.

The DRS->PVS translator generates a PVS function corresponding to the state to state transition of the DRS model. PVS was used to analyze the functional properties of the specification. For example, we show that the `trdy_lo_o` signal is asserted only when `t_abort` is false and `ready` is true with the PVS theorem:

```
 trdy_on_write: THEOREM
        (FORALL (t_abort: bit, tar_dly: bit, ready: bit):
           compute_trdy_lo(write, t_abort, tar_dly, ready) = true_lo
                IFF  NOT(t_abort) AND ready).
```

The `From_idle_goto_busy` theorem states that from IDLE, only when `frame_lo_i` is asserted, the Target sequencer goes to the BUS BUSY state.

```
From_idle_goto_busy: THEOREM
  (FORALL ((frame_lo_i: bit), (irdy_lo_i: bit), (trdy_lo_i: bit),
           (stop_lo_i: bit), (perr_lo_i: bit), (serr_lo_i: bit),
           (devsel_lo_i: bit), (ready: bit), (t_abort: bit),
           (term: bit), (state: state_type),
           (cbe_reg: [bit, bit, bit, bit]), (tar_dly: bit),
           (par_dat: bit), (par_en: bit), (par_i: bit),
           (perr_dat: bit), (r_perr: bit), (rperr_reg: bit)):
              idle(frame_lo_i, irdy_lo_i, trdy_lo_i, stop_lo_i,
                   perr_lo_i, serr_lo_i, devsel_lo_i, ready,
                   t_abort, term, state, cbe_reg, tar_dly, par_dat,
                   par_en, par_i, perr_dat, r_perr, rperr_reg)
               = bus_busy
               IFF (frame_lo_i = true_lo))
```

Many of the functional properties verified in PVS were also verified in the Verysys model checker. Both PVS and Verysys were useful in finding errors in the design. Early in the design process, we used sample equations from the PCI specification as a guide to developing the DRS specification. PVS uncovered overlaps in some of the equations. A set of conditions would satisfy two different equations.

## 5.2 Formal Synthesis using DRS

In the second stage, formal synthesis is used to manipulate the design hierarchy and derive a VHDL description from the top-level behavior specification. This process requires manual guidance from the designer. DRS provides automated support for transforming the specification to a concrete implementation, however, design decisions are made by the designer. DRS maintains correctness and does not allow the introduction of errors. The key benefit is that it provides the designer with direct control over the synthesis process.

DRS can manipulate a large class of designs including datapath and/or control oriented circuits. The PCI specification is a control-dominated circuit geared for bus protocol. DRS allowed us to manipulate the PCI design hierarchy providing a means of managing the complexity of the verification and defining the synthesized VHDL modules. We found that manipulating the design hierarchy of the VHDL would impact how the VHDL compiler would synthesize the design. Hierarchy played an important role in the speed of the synthesized circuit. The synthesizer did better when the design was in logically organized major blocks than a totally flat description or when there were many small modules instantiated in the larger ones.

The derivation was limited to obtaining a structural specification and generating the support modules from DRS libraries. We added four valued logic libraries to DRS. This enabled DRS to generate tristated input/output signals which are essential in a bus implementation.

The following table summaries the number of derivation steps, the specification and implementation size for each of the modules, along with the top-level mchip module.

| | Add4 | Dec4 | Backend | Txface | Mxface | Tconf | Tlock | Lock | mchip |
|---|---|---|---|---|---|---|---|---|---|
| DervSteps | 14 | 14 | 15 | 128 | 77 | 30 | 19 | 9 | 55 |
| Spec Size | 899 | 899 | 4440 | 12800 | 13906 | 5507 | 732 | 347 | 6209 |
| Imp. Size | 3194 | 3434 | 14286 | 12554 | 7471 | 10632 | 563 | 416 | 55790 |
| VHDL Size | 2669 | 2849 | 11909 | 11832 | 8425 | 8792 | 1002 | 858 | 48973 |
| VHDL Comp | 2669 | 2849 | 5493 | 9163 | 8425 | 8792 | 1002 | 858 | 9722 |

DRS and VHDL sizes include all the modules that make up the component. The component VHDL size lists only the size of that component. All sizes in bytes.

## 5.3 VHDL Generation, Validation and Synthesis

### 5.3.1 VHDL Generation

Once the design is refined to a concrete architecture in DRS, VHDL files are automatically generated and the VHDL Validation and Synthesis process begins. Model Technologies ModelSIM is used to simulate the VHDL. To streamline our simulation environment, we created interfaces from the DRS simulator to the VHDL and netlist simulator. This provided us the ability to localize our test vector generation within the DRS framework, and then automatically generate test vectors to validate the netlist generated by the VHDL compiler, and VHDL simulator.

The tools we used understood only a restricted subset of the VHDL language. We had to tune the VHDL generation toward the common syntax used among these tools. For example, the Verysys VHDL type checker could not resolve predicates of the form: `ad(1:0) = "00".` The DRS VHDL generation had to produce expressions of the form: `ad(1) = '0' and ad(0) = '0'.`

The VHDL compiler infers registers in a design depending on the way the code is written. Rather than an implicit mechanism to infer registers, we controlled the introduction of registers in the design by an explicit register entity, that served as a state holding abstraction and directly corresponded to DRS registers. The combinational logic is expressed as simple equations of assignments and entity instantiation. The resulting VHDL follows the intended implementation architecture closely.

To improve performance we experimented with several hierarchical design layouts. When flattening hierarchies the circuits were logically equivalent. However the circuit speed varied widely.

In generating VHDL, DRS constructs had to be mapped carefully over to VHDL constructs to ensure the semantics of the DRS expression is maintained. One problem we ran into was generating VHDL code for nested DRS if-then-else expressions. These expressions cannot be converted to selected signal assignments (WITH statement) unless the else branch guard is ANDed with the negated test expression. However, conditional signal assignment behaves just like a nested DRS if-then-else expression and is used instead of the WITH statement. In fact, the Verysys model checker uncovered this bug in the DRS VHDL generation.

## 5.3.2 VHDL Validation

The Verysys model checker is used to validate the VHDL against the PCI compliance model written in CIL. The underlying model checking technology used by the Verysys tools is the Siemens Circuit Verification Environment (CVE) [3]. The system is a BDD based symbolic model checker. It supports EDIF and VHDL, and generates VHDL test benches for counter examples.

Circuit properties are written in CIL (Circuit Interval Language). CIL formulae are built up from timed predicates that consist of a state predicate and a temporal specification. The temporal specification describes when the machine should be in a state that satisfies the state predicate. The state predicate is given in the subset of Boolean expressions in VHDL. The temporal specifications refer either to a particular point of time, or to a whole period. A point of time is specified after the keyword `at`. A period is specified by an interval, which is a uniform representation of three different types: `[t1, t2]`, refers to the time between `t1` and `t2` inclusively, `[t1, infinite]` , refers to t and every point after `t`, `[t, p]`, refers to the time between `t` and the last point of time before the state predicate `p` is satisfied for the next time.

An interval is preceded by `during` or `within` to specify whether the state predicate holds during the whole period or at least once in the interval. Times are either integer constants or defined relative to a variable `t` which is universally or existentially quantified by `always` and `finally`.

As an example, we express the property that the "Target Sequencer will never deadlock" as:

```
theorem target_deadlock;
  assume: (set = '0' during [0, infinite]);
  prove: always(possibly state = idle within [t, infinite]);
end theorem;
```

The assumption eliminates the reset state, and the proof guarantees that no matter what state the Target Sequencer is in, there exists a path to the idle state.

We prove that the Target Sequencer that implements the sustained tristate signals correctly with the following theorem:

```
theorem target_sustained_tristate_trdy;
  assume: (set = '0' during [0, infinite]);
  prove: always((trdy_lo_oe = '1' at t-1) and
                (trdy_lo_oe = '0' at t)
                   implies (trdy_lo_o = '1' at t-1));
end theorem;
```

In order for a signal to adhere to the sustained tristate property, it must drive the signal high one clock cycle before tristating the signal.

Most of the effort at this stage was spent developing the PCI compliance model. It was critical to be able to ask the "right" question. This was difficult since we had no prior understanding of the PCI protocol. Once the protocol was understood, writing the CIL properties from the PCI specification was fairly straight forward and the actual running of the model checker was automatic. Counter examples generated by the model checker were validated with the ModelSIM simulator at the VHDL level as well as in the DRS simulator. This capability allowed us to pinpoint if the problem was in the top-level DRS specification, VHDL generation, or VHDL code.

The design environment of this project consisted of two dynamic aspects: on the one hand the engineering process and on the other the formal process. From initial specification to working hardware the model checker did not find any errors that our hardware engineer did not find using traditional techniques. The model checking was lagging behind in this process. Errors uncovered by the engineering process led to revisions in the DRS specification.

After working hardware was achieved the model checker started finding errors in the design that the simulator did not uncover. This was due to three facts. First, the simulation tests were not exhaustive. Second, hardware and specification reached a level of maturity where the core appeared to work for most cases. Thirdly, we developed a better understanding of the PCI protocol.

The compliance model provides a comprehensive formal validation of PCI compliance and becomes extremely valuable in providing exhaustive analysis of the VHDL model. Inconsistencies found in the PCI specification were documented, and design decisions were made to resolve them.

#### 5.3.3 VHDL Synthesis

The VHDL files are input to Synopsys FPGA Express compiler for netlist synthesis. The issue in this process is that minor changes to the VHDL would result in significant performance changes in the synthesized netlist.

### 5.4 Netlist Validation and Mapping

The next stage involves simulating the netlist, and using the model checker to validate that the VHDL synthesis has not introduced any errors. Timing analysis is also done at this time. The netlist is then mapped to the appropriate target technology for hardware programming. At this stage, the logic netlist is validated using the Aldec netlist simulator. Test vectors written for the DRS architectural simulation are used at the VHDL and netlist level.

The logic netlist is formally verified using Verysys StructureProver. This ensures that the synthesized netlist behaves identical to the VHDL model in order to eliminate the possibility that logic bugs that would be introduced during VHDL synthesis. The equivalence checker compares the finite state machine models of the VHDL source and EDIF files of the synthesized netlist. There were no errors in the VHDL synthesis.

The Xilinx mapper then synthesized the appropriate configuration files for the target device.

### 5.5 Post-design Validation

Traditional hardware techniques were used for post-design validation.

The DRS Functional Test Environment (FTE) was used for hardware emulation of the synthesized PCI core. The FTE consists of the DRS simulation environment communicating with a Ampro EBX form factor Pentium based single board computer (SBC) and the PF2000 PC/104 FPGA module. The synthesized core is downloaded on to the PF2000 FPGA module. Then the DRS simulator drives the inputs of the circuit, single steps the clock, and samples the outputs, displaying them in the DRS simulator. In contrast to the functional simulation of the model in DRS, the FTE was used to compare the functional behavior of the model to that of a design that has been processed by implementation specific back end tools.

The core has been targeted to Xilinx XC4000 and Virtex family of FPGA devices. A working prototype is running in two different environments. The first system is a standard PCI/ISAbus AT motherboard with a AMD-K5 processor clocked at 133MHz. It includes an NE2000 compatible ISAbus based Ethernet card and a PCI VGA card.. The second system is an AMPRO PC/104+ system consisting of a Ampro EBX form factor Pentium based single board computer (SBC). Both systems are configured with 32Mb of memory and runs Linux RedHat 6.0, which is based on a 2.2.5 Linux kernel.

## 6 Conclusions

The methodology developed to build the FormalCORE PCI/32 is an example of how formal tools and traditional simulation and synthesis tools are integrated for the design and validation of VHDL IP cores. These cores can then be integrated into larger designs providing the building blocks for complex designs.

The FormalCORE PCI/32 and associated PCI compliance model consists of pre-designed, pre-verified VHDL components that can be integrated into larger designs and a validation suite providing exhaustive analysis of the VHDL models using a commercial model checker. The core has been designed to be flexible and can be adapted to a variety of designs with little or no modification to the VHDL or compliance model.

One observation is in the early stages of this project, traditional techniques led the design process. The ModelSIM VHDL simulator, Aldec netlist simulator, and hardware Logic Analyzer were used to debug the design. The model checker did not find any errors that either simulation or hardware debugging did not catch. The traditional techniques were satisfactory in achieving a working prototype. In the later stages of the project, the formal techniques led the design process. The model checker was able to find errors in the design that were not tested for in simulation. Using the DRS system, we were able to routinely make changes to the top-level specification, manipulate the design hierarchy, and re-synthesize the VHDL core. We could then re-validate the core against the compliance model automatically.

Both PVS and the Verysys model checker were useful in developing the PCI core. PVS was used to verify functional properties of the DRS top-level specification. Verysys was used to verify functional and temporal properties of the DRS generated VHDL. The Verysys verification effort was more extensive since the end goal was to develop a verified VHDL PCI core and compliance model.

This work has significantly enhanced our capability to design and validate VHDL cores. The enhancements added to the DRS system are general and can be used to synthesize a wide array of designs.

The future work on this topic is to extend the PCI core and Compliance model to the 64-bit PCI standard, retarget the core to operate at 66Mhz, and update the design to Revision 2.2 of
the PCI specification. In addition, we would like to perform an independent validation of the compliance properties.

## References

1. Bose, B. DRS – Derivational Reasoning System: A Digital Design Derivation System for Hardware Synthesis. In *Safety and Reliability in Emerging Control Technologies* (1996), S. Zaleswki, Ed., Elsevier.
2. Bose, B., Tuna, E., and Choppell, V., A Tutorial on Digital Design Derivation Using DRS., In *Formal Methods in Computer-Aided Design*, M. Srivas and A. Camilleri (eds.), Springer, 1996, pp. 270-274.
3. Bormann, J., Lohse, J., Payer, M., and Venzl, G., Model Checking in Industrial Hardware Design, In *Proceedings 32nd Design Automation Conference*, pp. 298-303, June 1995.
4. *IEEE Standard VHDL Language Reference Manual*. The Institute of Electrical and Electronics Engineers, Inc., New York, IEEE Std 1076-1993 edition, 1994.
5. Owre, S., Rushby, J., and Shankar, N. PVS: A Prototype Verification System. In *The 11th International Conference on Automated Deduction (CADE)*, Volume 607 of Lecture Notes in Artificial Intelligence (Saratoga, NY, June 1992), D. Kapur, Ed., Springer-Verlag, pp. 748-752.
6. PCI SIG. PCI Local Bus Specification, Revision 2.1., June 1995.
7. Shanley, T., and Anderson, D. *PCI System Architecture*. Addison Wesley, ISBM 0-201-40993-3.
8. Verysys Design Automation: Verysys Prover Environment Manual, VT-0050, Version 2.1, Rev. A, Verysys Design Automation, 42707 Lawrence Place, Fremont, CA 94538.
9. Xilinx, Foundation Series Software User Manual, Version 1.5i., Xilinx, 2100 Logic Drive, San Jose, CA 95124.